Border Case Testing of LLMs and their Extensions with Computationally Hard Queries

Anthony Butler[1], Iosif Itkin[2], Nikolai Dorfeev[2], and Rostislav Yavorskiy[2]

[1] Global Optima
Riyadh, Saudi Arabia
E-mail: abutler@globaloptima.net
[2] Exactpro Systems
27 Clements Lane, London, UK
E-mail: iosif.itkin@exactprosystems.com

Abstract. In this paper, we suggest an approach to test reasoning capabilities of large language models and their extensions. The method is based on the well-known software testing paradigm of border case analysis, when the system under test is prompted with two very close inputs that are to result in two completely different outputs. We use computationally hard logical problems, such as the boolean satisfiability and subset sum problem, to construct a parametric test suite. The results of the testing of several open LLMs are presented in the conclusion. The acquired dataset is publicly available.

Keywords: Software testing · Test generation · Large language models (LLMs) · Border case testing · Reasoning

1 Introduction

In modern agentic architectures, the reasoning and behavior of agents are often powered by large language models (LLMs), see [21, 23]. As many of us know, these models generate outputs autoregressively, which means they inherently introduce a small error probability at each step. While a 1% error rate might seem negligible, when an agent executes a sequence of actions, this inaccuracy compounds dramatically—a phenomenon known as error amplification.

LLMs generate text one token at a time [11], basing each new token on previous ones. This step-by-step process makes it practically impossible to achieve 100% accuracy across all tasks. Every step in a multi-step process carries a chance of error, and any mistake made early on can propagate through subsequent steps.

For instance, consider an agent with 99% per-step accuracy. In a process that requires 50 steps, even though each individual step is highly reliable, the overall process ends up being error-free only about 60% of the time. Now, contrast this with an agent that operates at 95% per-step accuracy—in that scenario, after 50 steps, the overall success rate drops dramatically to around 8%.

This means that a small difference in per-step accuracy (just 4 percentage points) can lead to a dramatic decline in overall performance when compounded

over many steps. This behavior is well known in fuzzy logic, when the t-norm for conjunction is defined as algebraic product, see e.g. [25].

The diagram 1 below illustrates the stark difference between these two scenarios:

- **Scenario 1: 99%** Agent starts with high per-step reliability, but after 50 steps, the compounded accuracy is around 60%.
- **Scenario 2: 95%** Agent exhibits a much more severe compounded accuracy decline, dropping overall accuracy to about 8% after 50 steps.

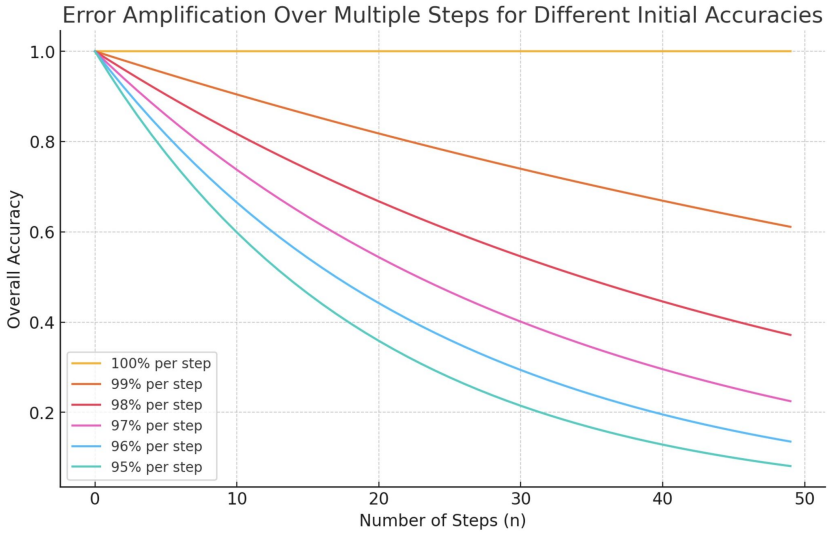

Fig. 1. The effect of decrease in accuracy for multi-step reasoning

This visual representation drives home the point that, in agentic systems, small inaccuracies at each step can add up quickly, potentially undermining the agent's ability to complete complex tasks. Therefore, understanding error amplification and developing approaches to mitigate it is crucial to designing robust agentic systems, particularly when they are used for complex tasks.

Some strategies include:

1. *Redundancy and verification.* Introduce self-reflection or external validation steps to catch and correct errors before they propagate [24].
2. *Uncertainty estimation.* Use confidence scoring to gauge the reliability of intermediate outputs and take corrective actions when needed [9].
3. *Hybrid approaches.* Combine multiple agents or use ensemble methods to cross-check results, reducing the impact of individual inaccuracies [19].

4. *Combination with formal proof systems.* Merging probabilistic reasoning with formal verification methods can provide a safety net to catch and correct errors before they lead to significant failures [16].

This is an emergent area, and it's likely that improvements will come not only from advancements in the models themselves but also from more sophisticated agent architectures that incorporate these mitigation strategies. In this paper, we address it from the perspective of software testing. The diagram 1 illustrates the effect of an unavoidable decrease in accuracy for multi-step reasoning. Our goal here is to measure the parameters of that decline for state-of-the-art LLMs.

The remainder of the paper is organized as follows. In section 2 we provide a brief introduction into the paradigm of model-based testing. Section 3 describes our idea of using computationally hard logical tasks to develop border case tests for LLMs. In section 4 that idea is converted into a specific test plan and test cases. Section 5 presents the test results.

2 Model based testing of AI systems

Model-based testing (MBT) is a systematic testing methodology in which test cases are derived from a formal representation — a model — of the system under test (SUT), see e.g. [7]. This approach is particularly relevant for complex systems, including large language models, AI-enhanced enterprise solutions, and other intelligent or adaptive systems, see the diagram on figure 2.

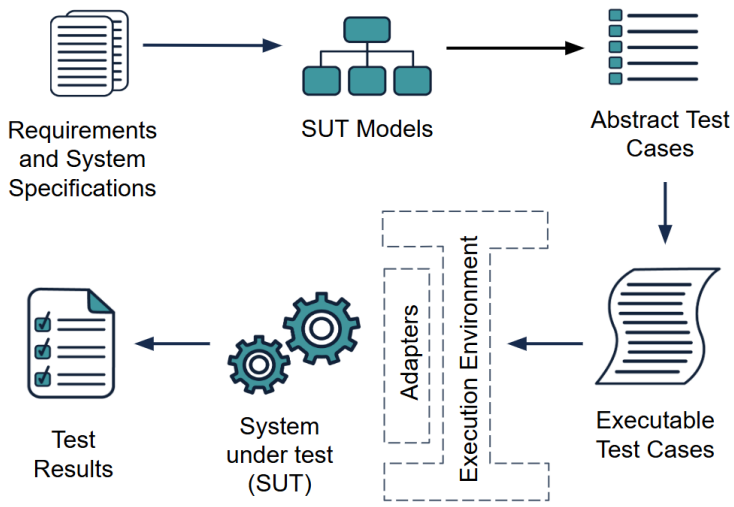

Fig. 2. Model based testing workflow

The evaluation of such systems must begin with a precise understanding of stakeholder expectations, as quality is ultimately defined by the degree to which

these expectations are met [17]. This initial phase involves thorough requirements analysis, serving as the foundation for constructing a corresponding model of the system. The model acts as a digital twin — a formal abstraction capturing key behaviors, states, and transitions of the system. This model can take various forms, including state machines [3], knowledge graphs [5], logical rules, or mathematical specifications [20], depending on the nature and complexity of the system. Test scenarios are systematically derived from the model through exploration and analysis. These scenarios are designed to ensure high coverage of the system's behavior and allow for rigorous examination of edge cases, failure modes, and specification boundaries [14]. Importantly, the model itself defines the expected system behavior, enabling automated or semi-automated verdicts on whether actual outputs conform to specified requirements [15]. Thus, model-based testing closes the verification feedback loop: the test results inform iterative improvements to the software, refinement of the model, and potential updates to the original requirements. This continuous integration of feedback enhances both the reliability and the maintainability of the system under test.

In order to test the reasoning capabilities of LLMs and their extensions, we use well-known formalized logical problems for the SUT models in the described workflow.

3 Border case test approach

Boundary testing is a technique used in software testing to check how a program behaves at the edges of valid input ranges, see e.g. [10, 13]. Instead of testing many different values, boundary testing focuses on the smallest and largest values that the program should accept, as well as just outside those limits. This is important because errors often occur at these boundary points, where the system's logic might behave unexpectedly. For example, if a program accepts numbers between 1 and 1000, boundary testing would check values like 0, 1, 1000, and 1001. This helps to find bugs more quickly and efficiently. It also reduces the number of test cases needed, as we are not testing every possible value. By targeting the most error-prone areas — the boundaries — this method saves time and increases the chances of catching mistakes early. In more general setting, boundary testing helps detect errors occurring at the boundary values of valid or invalid partitions. The partitioning involves dividing the input space into smaller components (equivalence classes) to make them more manageable.

In this paper, the idea of boundary testing is conducted in the following way. We start with an input query that is known to be computationally hard, for which the answer exists or does not exist depending on the input parameters, for example, the satisfiability problem; see e.g. [2, 8]. We ask SUT to find an interpretation that satisfies a given Boolean formula. Some formulas are satisfiable, so the answer could be found, and some other are not. Then, a border case test would consist of two formulas φ_1 and φ_2, such that:

1. One formula is satisfiable, and the other is not.
2. Syntactically, φ_1 and φ_2 are very similar (differ in just one symbol or so).

Another type of computationally hard problem we consider is the subset sum problem and its variations: there is a multiset $S = \{s_1, \ldots, s_n\}$ of integers and a target-sum T, and the question is to decide whether any subset of the integers sum to precisely T; see e.g. [8, 4, 18]. Similarly, a border case would consist of two problem occurrences with parameters S_1, T_1 and S_2, T_2 correspondingly. Such that:

1. For one configuration, the solution exists, and for the other one, it does not.
2. Syntactically, the two problem definitions with S_1, T_1 and S_2, T_2 are correspondingly very similar (differ by a single symbol or so).

4 Test plan

4.1 System under test (SUT)

For our computational experiments, we took three popular LLM services that provide API acceess:

– *Gemini,* a multimodal large language model developed by Google DeepMind, and the successor to LaMDA and PaLM-2 [22].
– *Mistral AI,* an open source large language model developed by a France-based artificial intelligence startup [12].
– *GPT-4,* a multimodal large language model trained and created by OpenAI and the fourth in its series of GPT foundation models [1].

4.2 Selected test cases

The test strategy is illustrated on diagram 3.

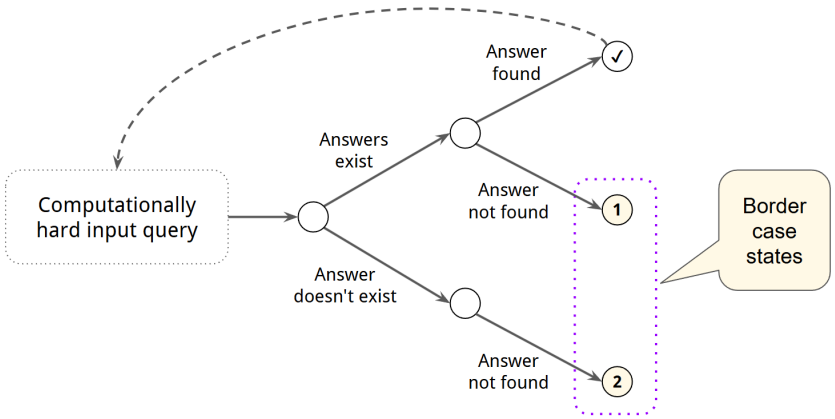

Fig. 3. LLMs reasoning capabilities testing strategy

We start with a formalized SUT model as a math problem solving machine, and construct several similar queries based on computationally hard logical problems. Since computational resources of the SUT are limited, the system will get into a border case state and will have to utilize its reasoning module.

An effective test case must be clear, relevant, and simple. In selecting test cases, we try to avoid unnecessary complexity, focusing only on the essential elements required to validate a particular behavior. That is necessary for enhancing the interpretability and reliability of testing outcomes. With that requirements in mind, the following parametric prompt was designed:

> I'm considering a business plan for a company that produces engines. First year production is planned to be T_1 engines. For each subsequent year the following options are possible:
>
> 1) Conservative year - the production will increase by R engines compared to the previous year.
>
> 2) Optimistic year - the number of the produced engines will increase by D_{opt} percent (rounded to the nearest integer).
>
> 3) Pessimistic year - the production will decrease by D_{pes} percent (rounded to the nearest integer).
>
> Each year starting from year 2 could be either conservative or optimistic, or pessimistic independently from the other years. Would it be possible in year N to get the production in interval between A and B? If yes, provide the scenario. If no, just say "No such scenario". If the solution is too difficult, just say "Answer unknown".

The prompt has seven parameters: T_1, R, D_{opt}, D_{pes}, N, A, and B, which are replaced by actual numbers before sending the query to SUT. We developed a set of representable problem variations depicted in the Table 1.

Table 1 represents a parametric test suite, containing 20 similar prompts to an LLM. The prompt has 7 parameters that are to be replaced by actual numbers from the table, which contains 10 pairs of configurations. In each pair the tests differ in one symbol only, yet one of the two queries has an answer, the other should result in empty set. The column `Oracle` contains pre-computed results that are used to check if the test has passed or failed.

Table 1. Summary of test configurations

TestID	T1	R	D_opt	D_pes	N	A	B	Oracle
1	3000	30	12%	15%	2	3200	3400	{3360}
2	3000	30	12%	15%	2	3200	3300	{}
3	3000	30	12%	15%	3	2600	2900	{2856}
4	3000	30	12%	15%	3	2600	2800	{}
5	3000	30	12%	15%	4	2600	2800	{2601, 2606, 2610}
6	3000	30	12%	15%	4	2700	2800	{}
7	3000	30	12%	15%	5	3000	3200	{3120}
8	3000	30	12%	15%	5	3000	3100	{}
9	3000	30	12%	15%	6	4000	4200	{4013}
10	3000	30	12%	15%	6	4100	4200	{}
11	3000	30	12%	15%	7	4400	4700	{4495}
12	3000	30	12%	15%	7	4500	4700	{}
13	3000	30	12%	15%	8	4100	4300	{4100, 4101, 4107}
14	3000	30	12%	15%	8	4200	4300	{}
15	3000	30	12%	15%	9	5600	5900	{5638, 5639}
16	3000	30	12%	15%	9	5700	5900	{}
17	3000	30	12%	15%	10	5700	6000	{5704}
18	3000	30	12%	15%	10	5800	6000	{}
19	3000	30	12%	15%	15	8700	8900	{8704, 8706, 8708, 8710, 8718, 8719, 8727, 8735}
20	3000	30	12%	15%	15	8800	8900	{}

5 Acquisition of Model Responses and Evaluation

Using API, we acquired formal responses from OpenAI, Gemini, and Mistral AI models. The versions are as follows:

- **Gemini**: `gemini-2.0-flash`
- **Mistral**: `mistral-large-latest` (as of 24.11)
- **OpenAI**: `gpt-4.1`

All responses were configured to be in JSON format with a description of the expected schema and validation of a response locally. Since the LLMs are non-deterministic, it makes sense to run each test several times. In our experiments, we submitted each query 50 times, and then computed the overall accuracy for each N. Although all models responded a bit differently each time, and sometimes their answer changed from correct to incorrect and vice versa. The acquired dataset is publicly available[3] [6].

Each LLM response consisted of the following three components:

1. **Satisfiability flag**: whether an appropriate solution exists for the given task (Yes/No),

[3] https://www.kaggle.com/datasets/d0rich/np-hard-problem-llms-responses/data

2. **Example actions**: an illustrative example of actions that constitute a valid solution (if applicable),
3. **Rationale**: free-form explanation of the model's reasoning.

For evaluation purposes, only the first two components were considered. The scoring was performed as follows:

- For tasks **with existing solutions**:
  - If the satisfiability flag was incorrect, the response received a score of 0.
  - If the satisfiability flag was correct, the example actions were checked against the known set of solutions:
    - If the example matched a valid solution, the response received a score of 1.
    - Otherwise, the response received a score of 0.
- For tasks **without existing solutions**:
  - If the satisfiability flag was incorrect, the response received a score of 0.
  - If the satisfiability flag was correct, the response received a score of 1.

General evaluation is depicted in Fig. 4.

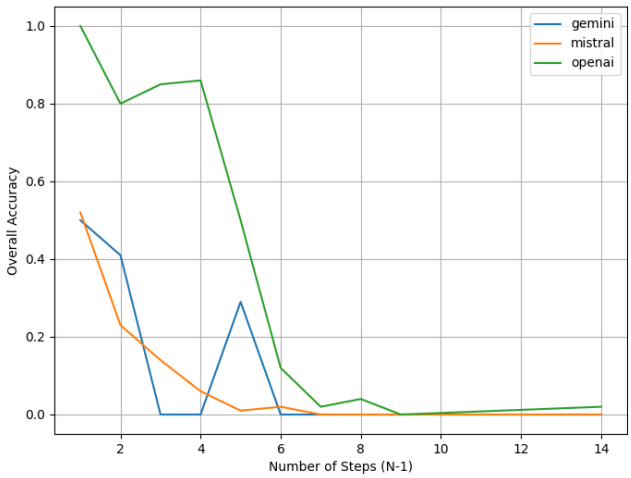

Fig. 4. Overall Testing Results

It might look like OpenAI model drastically outperforms all the other models, however Fig. 5 shows, that LLMs are comparable if a solution exists.

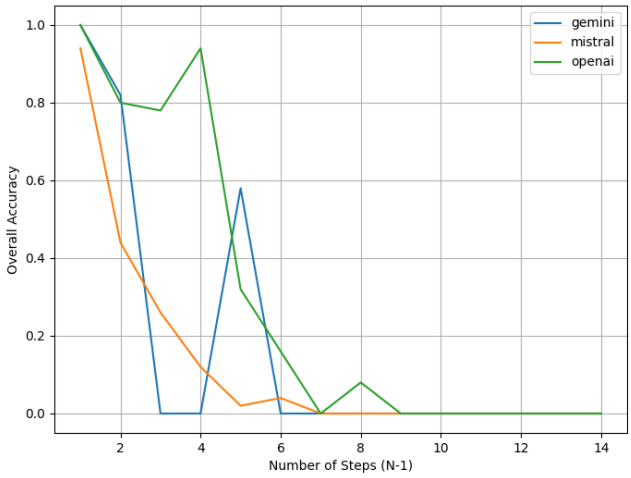

Fig. 5. Comparison Only for Problems with Existing Solution

On Fig. 6 we can see that most of the models are trying to give a positive response even if it does not exist.

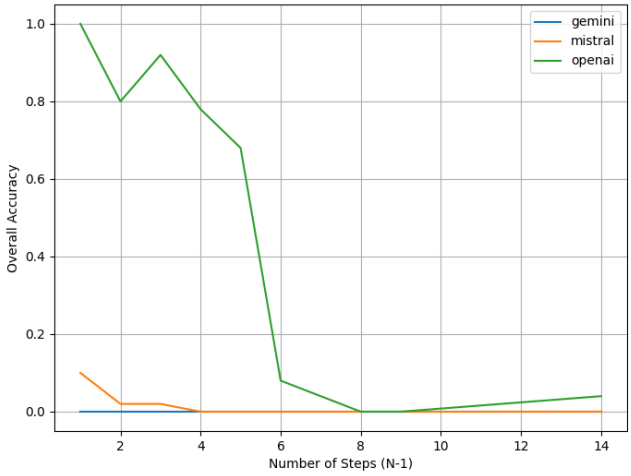

Fig. 6. Comparison Only for Problems without Existing Solution

It is worth noting, that sometimes in reasoning part model specifies, that the result is out of limits being the closest solution. We haven't used this information in evaluation.

6 Conclusion

In conclusion, None of the systems was able to build correct reasoning longer than 6 steps with at least 20% success rate — see the Fig. 4. On average, OpenAI outperformed the other implementations. One possible explanation is that OpenAI writes Python code and runs it under the hood. Because of this, the boost might be lost in solving more complex business tasks.

The fitted exponential decay model $Overall\ Accuracy(n) = a^n$ (Fig. 1) offers a compact way to compare reasoning stability across models. The parameter a reflects per-step retention, with higher values indicating better robustness. For OpenAI, the best fit was obtained with $a = 0.833$ (correlation = 0.917), suggesting relatively strong per-step stability. Gemini yielded $a = 0.537$ (correlation = 0.837), and Mistral, while having the lowest $a = 0.501$, achieved the best fit with a correlation of 0.997, indicating highly regular decay. These results capture both the reliability and predictability of each model under extended multi-step inference.

Redundancy and verification. Mechanisms such as prompting the model to reason before answering showed tangible benefits in our experiments. When models were guided to provide justifications first, their success rates improved. This aligns with the notion that internal self-checks or external validation steps can help catch and correct errors before they propagate.

Uncertainty estimation. This is critical for detecting when a model's output should not be trusted. In our results, models frequently produced confident but incorrect answers, especially on unsolvable test cases. If confidence scoring had been integrated, these false positives might have been flagged or avoided, reducing overconfident failure modes.

Hybrid approaches. All the models responded on their own in our experiments, but in a situation of product development we could use them together to be able to choose the best possible answer.

Formal proof systems. During the automatization of this study, we implemented a formal validation mechanism for the purpose of response validation. However, this type of mechanism can also be used with a LLM in order to improve system quality.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Blass, A., Gurevich, Y.: On the unique satisfiability problem. Information and Control **55**(1-3), 80–88 (1982)

3. Börger, E.: The abstract state machines method for high-level system design and analysis. In: Formal Methods: State of the Art and New Directions, pp. 79–116. Springer (2010)
4. Caprara, A., Kellerer, H., Pferschy, U.: The multiple subset sum problem. SIAM Journal on Optimization **11**(2), 308–319 (2000)
5. Chen, X., Jia, S., Xiang, Y.: A review: Knowledge reasoning over knowledge graph. Expert systems with applications **141**, 112948 (2020)
6. Dorofeev, N.: Np-hard problem llms responses. https://www.kaggle.com/datasets/d0rich/np-hard-problem-llms-responses/data (2025), accessed: 2025-05-06
7. El-Far, I.K., Whittaker, J.A.: Model-based software testing. Encyclopedia of Software Engineering (2002)
8. Garey, M.R., Johnson, D.S., et al.: A guide to the theory of np-completeness. Computers and intractability (1990)
9. He, J., Yu, L., Li, C., Yang, R., Chen, F., Li, K., Zhang, M., Lei, S., Zhang, X., Beigi, M., et al.: Survey of uncertainty estimation in large language models-sources, methods, applications, and challenge (2025)
10. Hoffman, D., Strooper, P., White, L.: Boundary values and automated component testing. Software Testing, Verification and Reliability **9**(1), 3–26 (1999)
11. Iqbal, T., Qureshi, S.: The survey: Text generation models in deep learning. Journal of King Saud University-Computer and Information Sciences **34**(6), 2515–2528 (2022)
12. Karamcheti, S., Orr, L., Bolton, J., Zhang, T., Goel, K., Narayan, A., Bommasani, R., Narayanan, D., Hashimoto, T., Jurafsky, D., et al.: Mistral–a journey towards reproducible language model training (2021)
13. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: 15th international symposium on software reliability engineering. pp. 139–150. IEEE (2004)
14. Lee, J., Kang, S., Jung, P.: Test coverage criteria for software product line testing: Systematic literature review. Information and Software Technology **122**, 106272 (2020)
15. Li, N., Offutt, J.: Test oracle strategies for model-based testing. IEEE Transactions on Software Engineering **43**(4), 372–395 (2016)
16. Lu, M., Delaware, B., Zhang, T.: Proof automation with large language models. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. pp. 1509–1520 (2024)
17. Madu, C.N.: Introduction to iso and iso quality standards. In: Handbook of total quality management, pp. 365–387. Springer (1999)
18. Martello, S., Toth, P.: A mixture of dynamic programming and branch-and-bound for the subset-sum problem. Management Science **30**(6), 765–771 (1984)
19. Schoenegger, P., Tuminauskaite, I., Park, P.S., Bastos, R.V.S., Tetlock, P.E.: Wisdom of the silicon crowd: Llm ensemble prediction capabilities rival human crowd accuracy. Science Advances **10**(45), eadp1528 (2024)
20. Spivey, J.M.: Understanding Z: a specification language and its formal semantics, vol. 3. Cambridge University Press (1988)
21. Sumers, T., Yao, S., Narasimhan, K., Griffiths, T.: Cognitive architectures for language agents. Transactions on Machine Learning Research (2023)
22. Team, G., Anil, R., Borgeaud, S., Alayrac, J.B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A.M., Hauth, A., Millican, K., et al.: Gemini: a family of highly capable multimodal models. arXiv preprint arXiv:2312.11805 (2023)

23. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al.: The rise and potential of large language model based agents: A survey. Science China Information Sciences **68**(2), 121101 (2025)
24. Xu, T., Wu, S., Diao, S., Liu, X., Wang, X., Chen, Y., Gao, J.: Sayself: Teaching llms to express confidence with self-reflective rationales. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. pp. 5985–5998 (2024)
25. Yen, J.: Fuzzy logic-a modern perspective. IEEE transactions on knowledge and data engineering **11**(1), 153–165 (1999)